

Problem M3.1: Out-of-order Scheduling

Ben Bitdiddle is adding a floating-point unit to the basic MIPS pipeline. He has patterned the design after the IBM 360/91's floating-point unit. His FPU has one adder, one multiplier, and one load/store unit. The *adder* has a four-cycle latency and is fully pipelined. The *multiplier* has a fifteen-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for write-back for loads) and that we have perfect branch prediction.

There are 4 floating-point registers, **F0–F3**. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (and loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

Ben is now deciding whether to go with (a) in-order issue using a scoreboard, (b) out-of-order issue, or (c) out-of-order issue with register renaming. His favorite benchmark is this DAXPY loop central to Gaussian elimination (Hennessy and Patterson, 291). The following code implements the operation $Y=aX+Y$ for a vector of length 100. Initially R1 contains the base address for X, R2 contains the base address for Y, and F0 contains a. Your job is to evaluate the performance of the three scheduling alternatives on this loop.

```
loop:
I1      L.D      F2, 0(R1)      ;load X(i)
I2      MUL.D   F1, F2, F0      ;multiply a*X(i)
I3      L.D      F3, 0(R2)      ;load Y(i)
I4      ADD.D   F3, F1, F3      ;add a*X(i)+Y(i)
I5      S.D      F3, 0(R2)      ;store Y(i)
I6      DADDUI  R1, R1, 8        ;increment X index
I7      DADDUI  R2, R2, 8        ;increment Y index
I8      DSGTUI  R3, R1, 800     ;test if done
I9      BEQZ    R3, loop        ;loop if not done
```

Problem M3.1.A

In-order using a scoreboard

Fill in the scoreboard in table M3.1-1 to simulate the execution of one iteration of the loop for in-order issue using a scoreboard (refer to slide L11-29 in 2005). Keep in mind that, in this scheme, no instruction is issued that has a WAW hazard with any previous instruction that has not written back (just like in the lecture slides). Recall the WB stage is only relevant for FP instructions (integer instructions can forward results). You may use ellipses in the table to represent the passage of time (to compress repetitive lines).

In steady state, how many cycles does each iteration of the loop take? What is the bottleneck?

Problem M3.1.B

Out-of-order

Now consider a single-issue out-of-order implementation (refer to slide L12-5 in 2005). In this scheme, the issue stage buffer holds multiple instructions waiting to issue. The decode stage can add up to one instruction per cycle to the issue buffer. The decode stage adds an instruction to the issue buffer if there is space and if the instruction does not have a WAR hazard with any previous instruction that has not issued or a WAW hazard with any previous instruction that has not written back. Assume you have an infinite issue buffer. Assume only one instruction can be dispatched from the issue buffer at a time.

Table M3.1-2 represents the execution of one iteration of the loop *in steady state*. Fill in the cycle numbers for the cycles at which each instruction issues and writes back. The first row has been filled out for you already; please complete the rest of the table. Note that the order of instructions listed is not necessarily the issue order. We define cycle 0 as the time at which instruction I_1 is issued.

Draw arrows for the RAW, WAR, and WAW dependencies that are involved in the *critical path* of the loop in table M3.1-2. In steady state, how many cycles does each iteration of the loop take?

Problem M3.1.C

Register Renaming

The number of registers specified in an ISA limits the maximum number of instructions that can be in the pipeline. This question studies register renaming to solve this problem. In this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers.

Table M3.1-3 shows instructions from our benchmark for two iterations using the same format as in Table M3.1-2. First, fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name. You may find it helpful to create a rename table. Rename both integer and floating-point instructions.

Next, fill in the cycle numbers for the cycles at which each instruction issues and writes back. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB). Assume that instruction I_2 was decoded in cycle 0, and cannot be issued until cycle 2. Also assume that you have an infinite ROB.

In steady state, how many cycles does each iteration of the loop take? What is the performance bottleneck?

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I ₁	-1	0	1	L.D	F2	R1	
I ₂				MUL.D	F1	F2	F0
I ₃				L.D	F3	R2	
I ₄				ADD.D	F3	F1	F3
I ₅				S.D		R2	F3
I ₆				DADDUI	R1	R1	
I ₇				DADDUI	R2	R2	
I ₈				DSGTUI	R3	R1	
I ₉				BEQZ		R3	

Table M3.1-2

	Time			Op	Dest	Src1	Src2
	Decode → Issue	Issued	WB				
I ₁	-1	0	1	L . D	T0	R1	
I ₂				MUL . D	T1	t0	F0
I ₃				L . D	T2	R2	
I ₄				ADD . D	T3		
I ₅				S . D			
I ₆				DADDUI			
I ₇				DADDUI			
I ₈				DSGTUI			
I ₉				BEQZ			
I ₁				L . D			
I ₂				MUL . D			
I ₃				L . D			
I ₄				ADD . D			
I ₅				S . D			
I ₆				DADDUI			
I ₇				DADDUI			
I ₈				DSGTUI			
I ₉				BEQZ			

Table M3.1-3

Problem M3.2: Out-of-Order Scheduling

This problem deals with an out-of-order single-issue processor that is based on the basic MIPS pipeline and has floating-point units. The FPU has one adder, one multiplier, and one load/store unit. The adder has a two-cycle latency and is fully pipelined. The multiplier has a ten-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for write-back for loads).

There are 4 floating-point registers, F_0-F_3 . These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (including loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB).

The instructions are committed in order and only one instruction may be committed per cycle. The earliest time an instruction can be committed is one cycle after write back.

For the following questions, we will evaluate the performance of the code segment in Figure M3.2-A.

I_1	L.D	$F_1, 5 (R_2)$
I_2	MUL.D	F_2, F_1, F_0
I_3	ADD.D	F_3, F_2, F_0
I_4	ADDI	$R_2, R_2, 8$
I_5	L.D	$F_1, 5 (R_2)$
I_6	MUL.D	F_2, F_1, F_1
I_7	ADD.D	F_2, F_2, F_3

Figure M3.2-A

Problem M3.2.A

For this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers. Assume that you have an **infinite ROB**.

Your job is to complete Table M3.2-1. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable. Since we have an infinite supply of register names, you should use a new register name each time a register is written (T0, T1, T2, ... etc). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register need to refer instead to the new register name.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I ₁	-1	0	1	2	L . D	T0	R2	-
I ₂	0	2	12	13	MUL . D	T1	T0	F0
I ₃	1				ADD . D			
I ₄					ADDI			-
I ₅					L . D			-
I ₆					MUL . D			
I ₇					ADD . D			

Table M3.2-1

Problem M3.2.B

For this question, assume that you have a **two-entry ROB**. An ROB entry can be reused **one cycle** after the instruction using it commits.

Your job is to complete Table M3.2-2. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new register names for each instruction, where applicable.

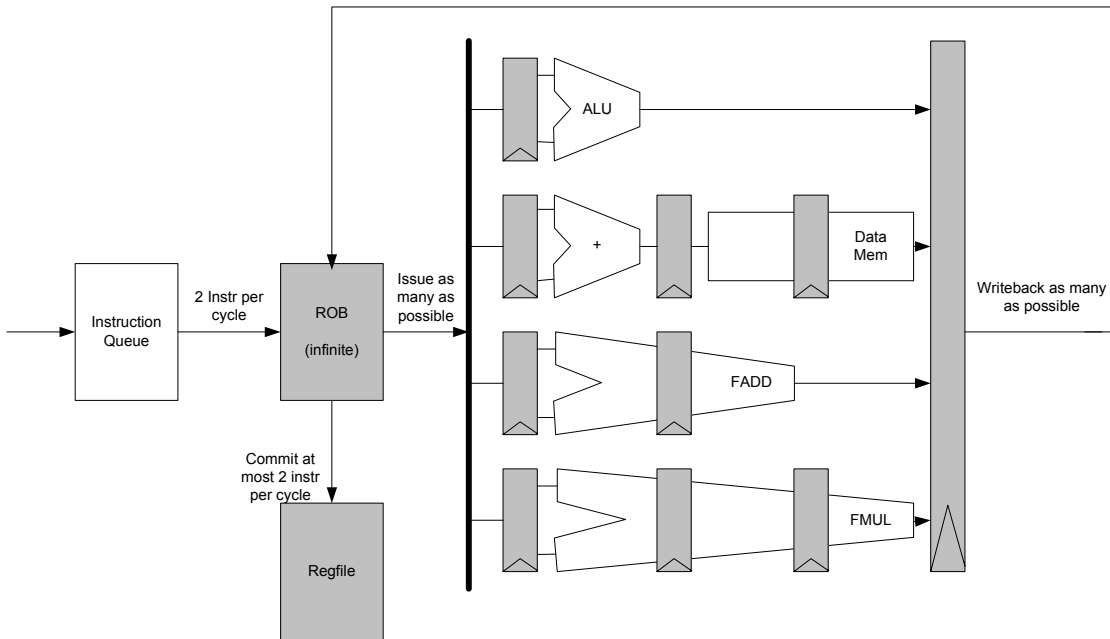
	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I ₁	-1	0	1	2	L . D	T0	R2	-
I ₂	0	2	12	13	MUL . D	T1	T0	F0
I ₃	3				ADD . D			
I ₄					ADDI			-
I ₅					L . D			-
I ₆					MUL . D			
I ₇					ADD . D			

Table M3.2-2

Problem M3.3: Superscalar Processor

Consider the out-of-order, superscalar CPU shown in the diagram. It has the following features:

- Four fully-pipelined functional units: ALU, MEM, FADD, FMUL
- Instruction Fetch and Decode Unit that renames and sends 2 instructions per cycle to the ROB (assume perfect branch prediction and no cache misses)
- An unbounded length Reorder Buffer that can perform the following operations on every cycle:
 - Accept two instructions from the Instruction Fetch and Decode Unit
 - Dispatch an instruction to each functional unit including Data Memory
 - Let Writeback update an unlimited number of entries
 - Commit up to 2 instructions in-order
- There is no bypassing or short circuiting. For example, data entering the ROB cannot be passed on to the functional units or committed in the same cycle.



Now consider the execution of the following program on this machine using:

```

I1      loop:   LD F2, 0(R2)
I2      LD F3, 0(R3)
I3      FMUL F4, F2, F3
I4      LD F2, 4(R2)
I5      LD F3, 4(R3)
I6      FMUL F5, F2, F3
I7      FMUL F6, F4, F5
I8      FADD F4, F4, F5
I9      FMUL F6, F4, F5
I10     FADD F1, F1, F6
I11     ADD R2, R2, 8
I12     ADD R3, R3, 8
I13     ADD R4, R4, -1
I14     BNEZ R4, loop

```

Problem M3.3.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10. Tags should not be reused.

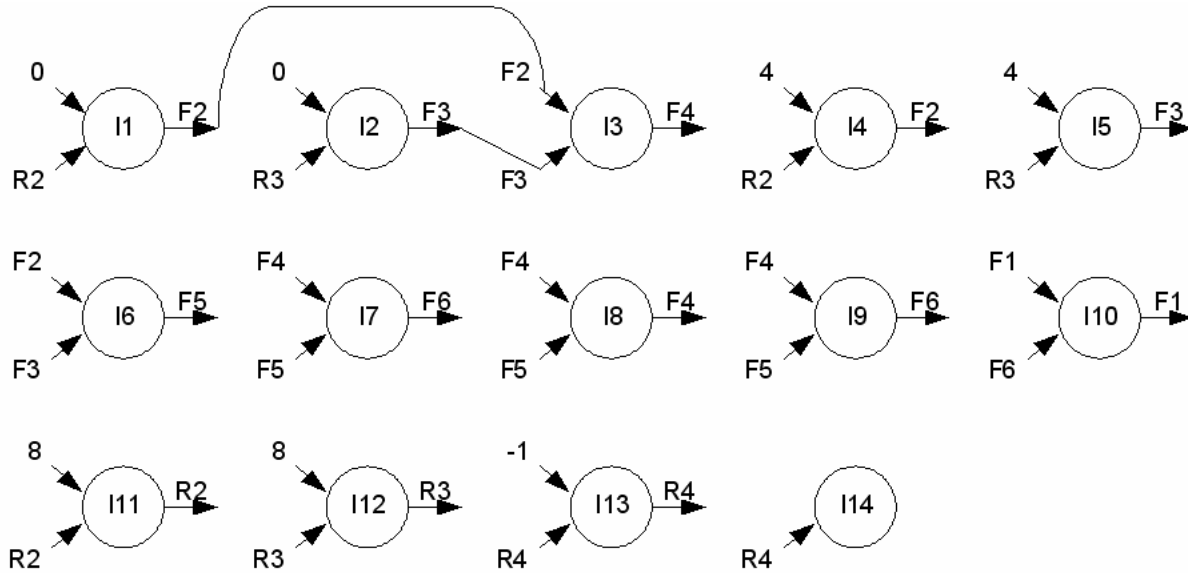
Instr #	Instruction	Dest	Src1	Src2
I1	LD F2, 0(R2)	T1	R2	0
I2	LD F3, 0(R3)	T2	R3	0
I3	FMUL F4, F2, F3			
I4	LD F2, 4(R2)		R2	4
I5	LD F3, 4(R3)		R3	4
I6	FMUL F5, F2, F3			
I7	FMUL F6, F4, F5			
I8	FADD F4, F4, F5			
I9	FMUL F6, F4, F5			
I10	FADD F1, F1, F6		F1	

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
R2										
R3										
F1										
F2	T1									
F3		T2								
F4										
F5										
F6										

Problem M3.3.B

Consider the execution of *one* iteration of the loop (I1 to I14). In the following diagram draw the data dependencies between the instructions after register renaming



Problem M3.3.C

The attached table is a data structure to record the times when some activity takes place in the ROB. For example, one column records the time when an instruction enters ROB, while the last two columns record, respectively, the time when an instruction is dispatched to the FU's and the time when results are written back to the ROB. This data structure has been designed to test your understanding of how a Superscalar machine functions.

Fill in the blanks in last two columns up to slot T13 (You may use the source columns for book keeping – no credit will be taken off for the wrong entries there).

Slot	Instruction	Cycle instruction entered ROB	Argument 1		Argument 2		dst	<i>Cycle dispatched</i>	<i>Cycle written back to ROB</i>
			src1	cycle available	Src2	cycle available	dst reg		
T1	LD F2, 0(R2)	1	C	1	R2	1	F2	2	6
T2	LD F3, 0(R3)	1	C	1	R3	1	F3	3	7
T3	FMUL F4, F2, F3	2			F3	7	F4		
T4	LD F2, 4(R2)	2	C	2	R2		F2		
T5	LD F3, 4(R3)	3	C	3	R3		F3		
T6	FMUL F5, F2, F3	3					F5		
T7	FMUL F6, F4, F5	4					F6		
T8	FADD F4, F4, F5	4					F4		
T9	FMUL F6, F4, F5	5					F6		
T10	FADD F1, F1, F6	5					F1		
T11	ADD R2, R2, 8	6	R2	6	C	6	R2		
T12	ADD R3, R3, 8	6	R3	6	C	6	R3		
T13	ADD R4, R4, -1	7	R4	7	C	7	R4		
T14	BNEZ R4, loop	7			C	Loop			
T15	LD F2, 0(R2)	8	C	8			F2	10	14
T16	LD F3, 0(R3)	8	C	8			F3	11	15
T17	FMUL F4, F2, F3	9					F4		
T18	LD F2, 4(R2)	9	C	9			F2		
T19	LD F3, 4(R3)	10	C	10			F3		
T20	FMUL F5, F2, F3	10					F5		
T21	FMUL F6, F4, F5	11					F6		
T22	FADD F4, F4, F5	11					F4		
T23	FMUL F6, F4, F5	12					F6		
T24	FADD F1, F1, F6	12					F1		
T25	ADD R2, R2, 8	13			C	13	R2		
T26	ADD R3, R3, 8	13			C	13	R3		
T27	ADD R4, R4, -1	14			C	14	R4		
T28	BNEZ R4, loop	14			C	Loop			

Problem M3.3.D

Identify the instructions along the longest latency path in completing this iteration of the loop (up to instruction 13). Suppose we consider an instruction to have executed when its result is available in the ROB. How many cycles does this iteration take to execute?

Problem M3.3.E

Do you expect the same behavior, i.e., the same dependencies and the same number of cycles, for the next iteration? (You may use the slots from T15 onwards in the attached diagram for bookkeeping to answer this question). Please give a simple reason why the behavior may repeat, or identify a resource bottleneck or dependency that may preclude the repetition of the behavior.

Problem M3.3.F

Can you improve the performance by adding at most one additional memory port and a FP Multiplier? Explain briefly.

Yes / No

Problem M3.3.G

What is the minimum number of cycles needed to execute a typical iteration of this loop if we keep the same latencies for all the units but are allowed to use as many FUs and memory ports and are allowed to fetch and commit as many instructions as we want.

Problem M3.4: Register Renaming and Static vs. Dynamic Scheduling

The following MIPS code calculates the floating-point expression $E = A * B + C * D$, where the addresses of A, B, C, D, and E are stored in R1, R2, R3, R4, and R5, respectively:

```
L.S      F0, 0(R1)
L.S      F1, 0(R2)
MUL.S    F0, F0, F1
L.S      F2, 0(R3)
L.S      F3, 0(R4)
MUL.S    F2, F2, F3
ADD.S    F0, F0, F2
S.S      F0, 0(R5)
```

Problem M3.4.A

Simple Pipeline

Calculate the number of cycles this code sequence would take to execute (i.e., the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive) on a simple in-order pipelined machine that has no bypassing. The datapath includes a load/store unit, a floating-point adder, and a floating-point multiplier. Assume that loads have a two-cycle latency, floating-point multiplication has a four-cycle latency and floating-point addition has a two-cycle latency. Write-back for floating-point registers takes one cycle. Also assume that all functional units are fully pipelined and ignore any write back conflicts. Give the number of cycles between the issue of the first load instruction and the issue of the final store, inclusive.

Problem M3.4.B

Static Scheduling

Reorder the instructions in the code sequence to minimize the execution time. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the simple in-order pipeline.

Problem M3.4.C

Fewer Registers

Rewrite the code sequence, but now using only two floating-point registers. Optimize for minimum run-time. You may need to use temporary memory locations to hold intermediate values (this process is called register-spilling when done by a compiler). List the code sequence and give the number of cycles this takes to execute.

Problem M3.4.D

Register renaming and dynamic scheduling

Simulate the effect of running the original code on a single-issue machine with register renaming and out-of-order issue. Ignore structural hazards apart from the single instruction decode per cycle. Show how the code is executed and give the number of cycles required. Compare it with results from optimized execution in M3.4.B.

Problem M3.4.E

Effect of Register Spills

Now simulate the effect of running code you wrote in M3.4.C on the single-issue machine with register renaming and out-of-order issue from M3.4.D. Compare the number of cycles required to execute the program. What are the differences in the program and/or architecture that change the number of cycles required to execute the program? You should assume that all load instructions before a store must issue before the store is issued, and load instructions after a store must wait for the store to issue.

Problem M3.5: Register Renaming Schemes

This problem requires the knowledge of Handout #10 (Out-of-Order Execution with ROB) and Lecture 11 and 12 (2005). Please, read these materials before answering the following questions.

Future File Scheme

In order to eliminate the step of reading operands from the reorder buffer in the decode stage, we can insert a second register file into the processor shown in Figure H10-A, called a *future file*. The future file contains the most up-to-date speculatively-executed value for a register, while the primary register file contains committed values. Each entry in the future file has a valid bit. A summary of the operation is given below.

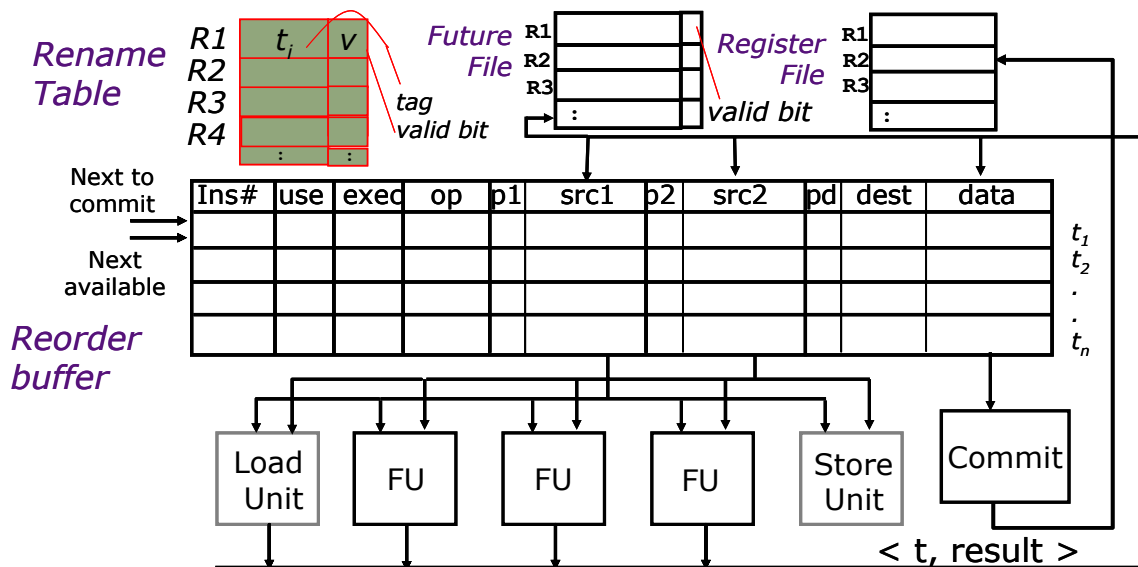


Figure M3.5-B

Only the decode and write-back stages have to change from the baseline implementation in Handout #10 to implement the future file, as described below:

Decode: Rename table, register file, and future file are read simultaneously. If the rename table has the valid bit set for an operand, then the value has not yet been produced and the tag will be used. Otherwise, if the future file has a valid bit set for its entry, then use the future file value. Otherwise, use the register file value. The instruction is assigned a slot in the ROB (the ROB index is this instruction's tag). If the instruction writes a register, its tag is written to the destination register entry in the rename table.

Write-Back: When an instruction completes execution, the result, if any, will be written back to the *data* field in the reorder buffer and the *pd* bit will be set. Additionally, any dependent instructions in the reorder buffer will receive the value. If the tag in the rename table for this register matches the tag of the result, the future file is written with the value and the valid bit on the rename table entry is cleared.

Problem M3.5.A

Finding Operands: Original ROB scheme

Consider the original ROB scheme in Handout #10, and suppose the processor state is as given in Figure H10-A. Assume that the following three instructions enter the ROB simultaneously in a single cycle, and that no instructions commit or complete execution in this cycle. In the table below, write the contents of each instruction's source operand entries (either the register value or a tag, t_1 , t_2 , etc., for both Src1 and Src2) and whether that entry came from the register file, the reorder buffer, the rename table, or the instruction itself.

Instruction	Src1 value	Regfile, ROB, rename table, or instruction?	Src2 value	Regfile, ROB, rename table, or instruction?
sub r5, r1, r3				
addi r6, r2, 4				
andi r7, r4, 3				

Problem M3.5.B

Finding Operands: Future File Scheme

In the future file scheme, explain why an instruction entering the ROB will never need to fetch either of its operands from the ROB.

Problem M3.5.C

Future File Operation

Describe a situation in which an instruction result is written to the ROB but might not be written to the future file. Provide a simple code sequence to illustrate your answer.

Problem M3.5.D

Handling Branch Mispredictions

In the original ROB scheme, a branch misprediction caused all instructions after the branch to be flushed from the ROB. Consider the following instruction sequence.

```
ADD R1, R2, R3
SUB R4, R5, R6
BEQ R7, R8, L1      # Taken branch.
XOR R9, R10, R11
ADD R1, R5, R9
```

Assume that there are no delay slots in the ISA and that the branch is incorrectly predicted not-taken. **In the future file scheme**, if all of the above instructions complete execution before the branch misprediction is detected, explain why simply flushing the mispredicted instructions is not sufficient for correct execution. What should be done instead?

Problem M3.6: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program:

```
loop:  
    LW    R4, 0(R3)  
    ADDI  R3, R3, 4  
    SUBI  R1, R1, 1  
b1:  
    BEQZ  R4, b2  
    ADDI  R2, R2, 1  
b2:  
    BNEZ  R1, loop
```

Assume the initial value of R1 is n ($n > 0$).

Assume the initial value of R2 is 0 (R2 holds the result of the program).

Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in lecture. We will be using a 2-bit predictor state machine, as shown below.

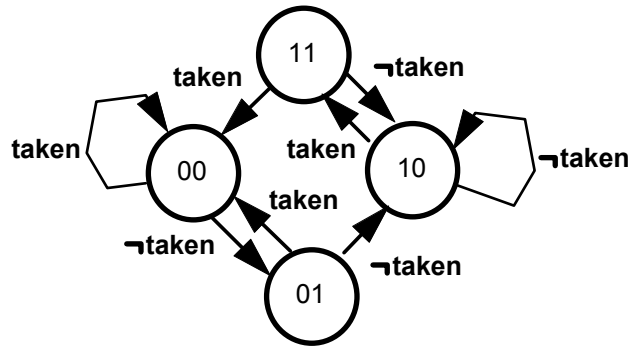


Figure M3.6-A. BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem M3.6.A

Program

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem M3.6.B

2-bit branch prediction

Now we will investigate how well our standard 2-bit branch predictor performs. Assume the inputs to the program are $n=8$ and $p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0, \dots$ etc.; i.e. the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table M3.6-1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table M3.6-1 contains an entry for every branch (either b1 or b2) that is executed. The Branch Predictor (BP) bits in the table are the bits from the BHT. For each branch, check the corresponding BP bits (indicated by the bold entries in the examples) to make a prediction, then update the BP bits in the following entry (indicated by the italic entries in the examples).

Problem M3.6.C

Branch prediction with one global history bit

Now we add a global history bit to the branch predictor, as described in lecture. Fill out Table M3.6-2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem M3.6.D

Branch prediction with two global history bits

Now we add a second global history bit. Fill out Table M3.6-3. Again, compute the number of mispredicts you get for the same input.

Problem M3.6.E

Analysis I

Compare your results from problems M3.6.B, M3.6.C, and M3.6.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For large n , what prediction scheme will work best? Explain briefly.

Problem M3.6.F

Analysis II

The input we worked with in this problem is quite regular. How would you expect things to change if the input were random (each array element were equally probable 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n ?

What does this tell you about when additional history bits are useful and when they hurt you?

System State		Branch Predictor		Branch Behavior	
PC	R3/R4	b1 bits	b2 bits	Predicted	Actual
b1	4/1	10	10	N	N
b2	4/1	10	10	N	T
b1	8/0	10	11	N	T
b2	8/0	11	11	N	T
b1	12/1	11	00		
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table M3.6-1

System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits		Predicted	Actual
			set 0	set 1	set 0	set 1		
b1	4/1	1	10	10	10	10	N	N
b2	4/1	0	10	10	10	10	N	T
b1	8/0	1	10	10	11	10		
b2	8/0							
b1	12/1							
b2	12/1							
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								

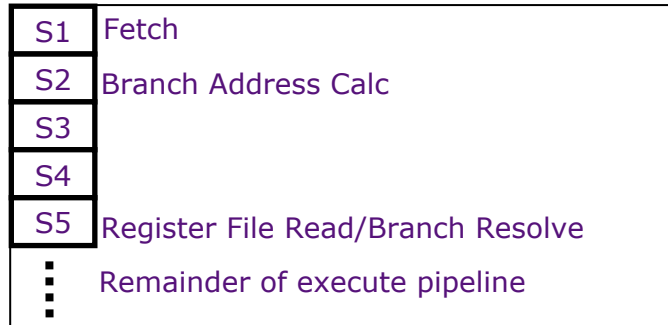
Table M3.6-2

System State			Branch Predictor								Behavior	
PC	R3/R4	history	b1 bits				b2 bits				Predicted	Actual
		bits	set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11		
b1	4/1	11	10	10	10	10	10	10	10	10	N	N
b2	4/1	01	10	10	10	10	10	10	10	10	N	T
b1	8/0	10	10	10	10	10	10	11	10	10		
b2	8/0											
b1	12/1											
b2	12/1											
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												

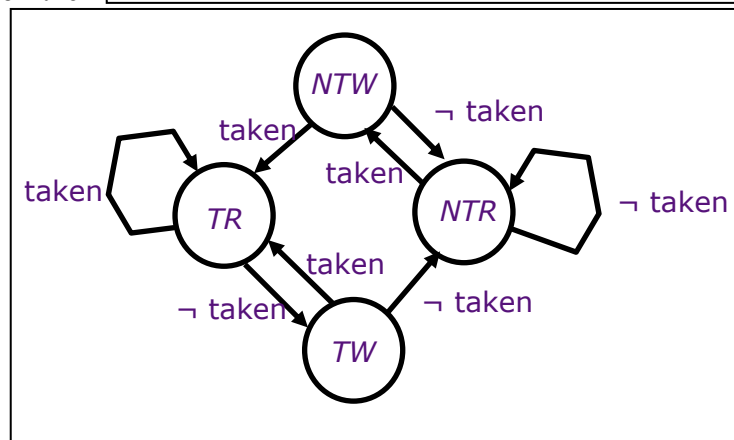
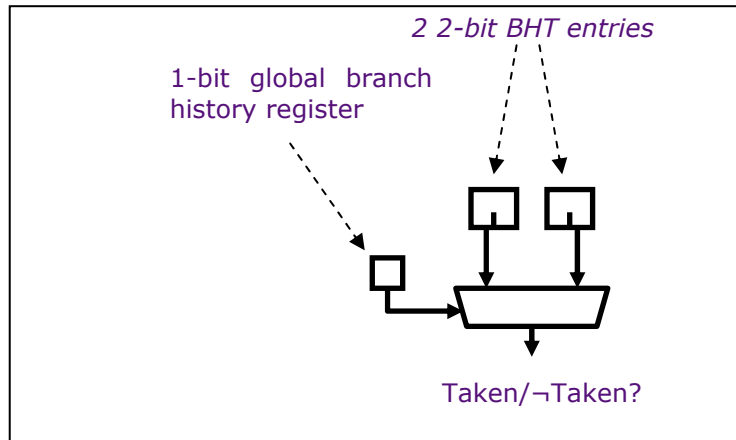
Table M3.6-3

Problem M3.7: Branch Prediction

Consider a CPU with a pipeline pictured on the right. The first stage of the pipeline fetches the instruction. The **second stage** of the pipeline recognizes branch instructions and **performs branch prediction** using a BHT. If the branch is predicted to be taken, it forwards the decoded target of the branch to the first stage, and kills the instruction in the first stage. The **fifth stage** of the pipeline reads the registers and **resolves the correct target of the branch**. If the branch target was mispredicted, the correct target is forwarded to the first stage, and all instructions inbetween are killed. The remainder of the stages finish the computation of the instruction.



The processor uses a **single global history bit** to remember whether the last branch was taken or not. There is only **one line in the BHT**, so the address of the branch instruction is not used for selecting the proper table entry. Each entry in the table is labeled as TW for Take Wrong, TR for Take Right, NTW for do Not Take Wrong and NTR for do Not Take Right, as pictured above. The setup of the BHT predictor is illustrated on the right.



In this question we will study execution of the following loop. This processor has **no** branch delay slots. You should assume that branch at address 1 is **never** taken, and that the branch at address 5 is **always** taken.

Instruction Label	Address	Instruction
LOOP	1	BEQ R2, R5, NEXT
	2	ADD R4, R4, 1
	3	MULT R3, R3, R4
NEXT	4	MULT R2, R2, 3847
	5	BNEZ R4, LOOP
	6	NOP
	7	NOP
	8	NOP
	9	NOP
	10	NOP

You should also **disregard any possible structural hazards**. The processor always runs at full speed, and there are **no pipeline bubbles** (except for those created by branches).

Problem M3.7.A

Now we study how well the history bit works, when it is being **updated by the fifth stage** of the processor. The fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in the table below.

You should fetch a new instruction on every cycle. You should fill in the *Branch Prediction* and *Prediction Correct?* columns for branch instructions only (note, the branch prediction actually happens one cycle **after** the instruction is fetched). You should fill in *Branch Predictor State* columns whenever they are updated. Please **circle instructions which will be committed**.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	1	T	N			
2	2					
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	2					
7	3					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						

Problem M3.7.B

Now we study how well the branch **history bit** works, when it **is being updated speculatively** by the second stage of the processor. If the branch is mispredicted, the fifth stage sets the branch history bit to the correct value. Finally, the fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in table below. The notation in the table should be same as in *Question M3.7.A*.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	1	T	N			
2	2			T		
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	2					
7	3					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						

Problem M3.8: Branch Prediction

In this question we explore the changes and impact of a branch predictor in the 5-stage pipeline.

Figures M3.8-A and M3.8-B are provided for the questions in this part (M3.8).

Figure M3.8-A shows the logic and state required to implement branch and jump instructions in the standard 5-stage pipeline. The design does not implement branch delay slots and assumes that branches are resolved in the decode stage. The diagram has been redrawn for clarity but is logically equivalent to the one used in Lecture 6 (2005).

Figure M3.8-B adds a PC predictor to the design. The predictor predicts the next PC based on the current PC. If in the Decode stage it is found that the prediction was incorrect, the fetch stage is flushed and the PC is overwritten with the correct Next PC. If the Decode stage finds that the prediction was correct, then the fetch stage continues predicting the next PC. Furthermore, we have provided some unspecified input wires to update the predictor, when needed.

The following questions are about the design in **Figure M3.8-B**.

Problem M3.8.A

What is the logic in the “Mispredict” block? (Write your answer as an equation, e.g. $(\text{NextPC} == 17) \& (\text{IR}_E == \text{BEQZ})$).

Mispredict =

Problem M3.8.B

What is the logic, if any, in the block that feeds the select of the IR_E register? You may use any of the labeled signals. Explain.

Problem M3.8.C

Is any change needed in the stall signal? Again you may use any of the labeled signals if needed to generate the new stall signal from the old one. Explain.

Problem M3.8.D

Does the processor function correctly if the PC Predict always mispredicts? Explain briefly. If your answer is “no” provide an example when it fails.

Problem M3.8.E

In this question we are concerned with the performance (i.e. cycle count to execute a program) compared to the pipeline that does not have the “PC Predict” block. Suppose the PC Predict always predicts “PC + 4”. The design performs identically in all cases except when the target is PC+4. Does the design with the PC Predict perform better or worse in the special case? Explain.

Better / Worse

Problem M3.8.F

Suppose the PC Predict can have maximum of 1K bits of storage and some small amount of logic. Give the best scheme you can design to implement PC Predict correctly. You will be judged on the quality of your solution, in addition to its correctness. Your solution must describe/discuss:

- The method for computing the predicted PC
- Common cases where it performs better than the simple scheme (i.e. “PC+4”)
- Common cases, if any, where it performs worse than the simple scheme (i.e. “PC+4”)
- The method for updating the prediction.

Problem M3.8.G

Can you think of any way to bring down the cost of executing an unconditional jump instruction to zero (not 1)?

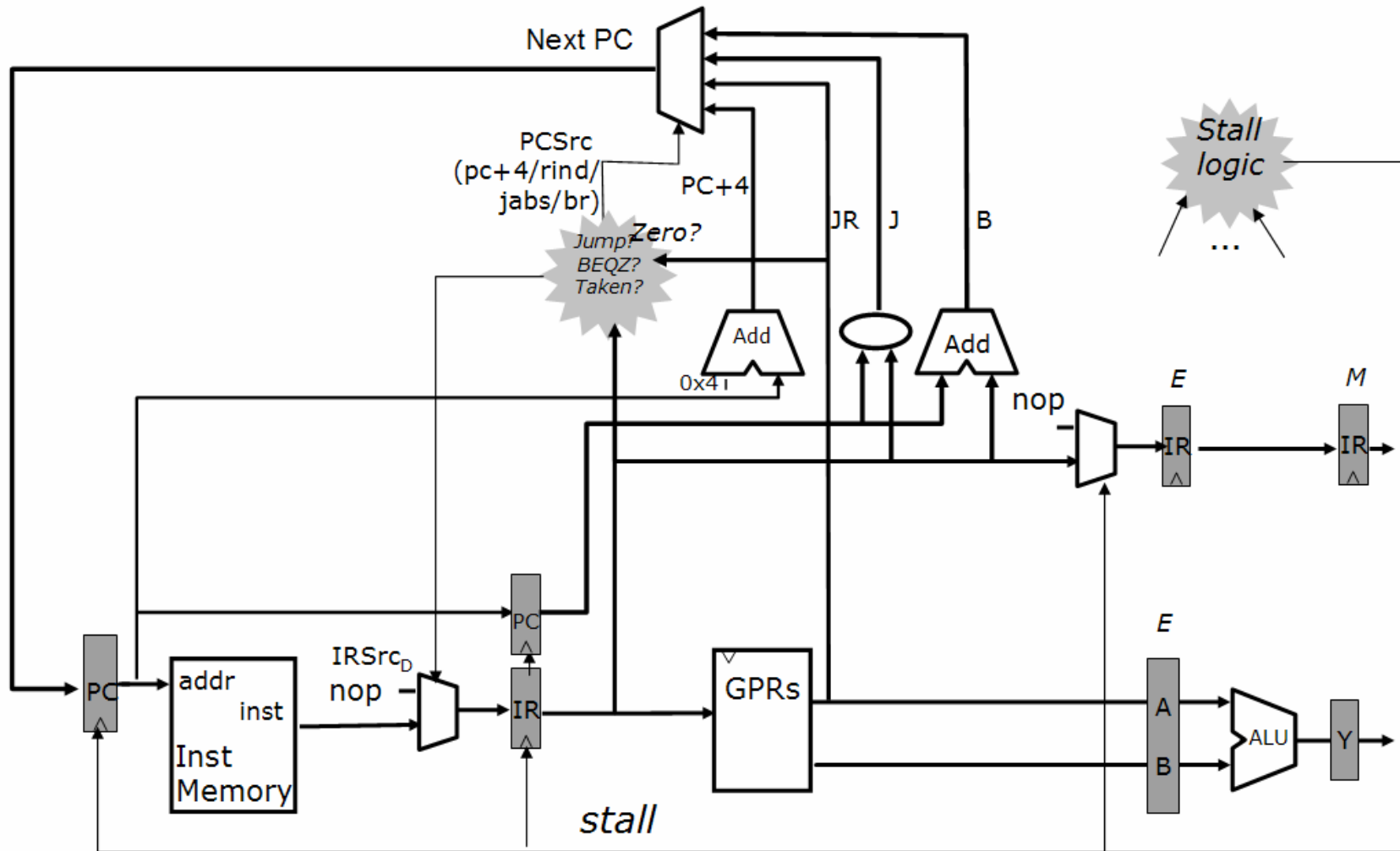


Figure M3.8-A. 5 stage MIPS pipeline

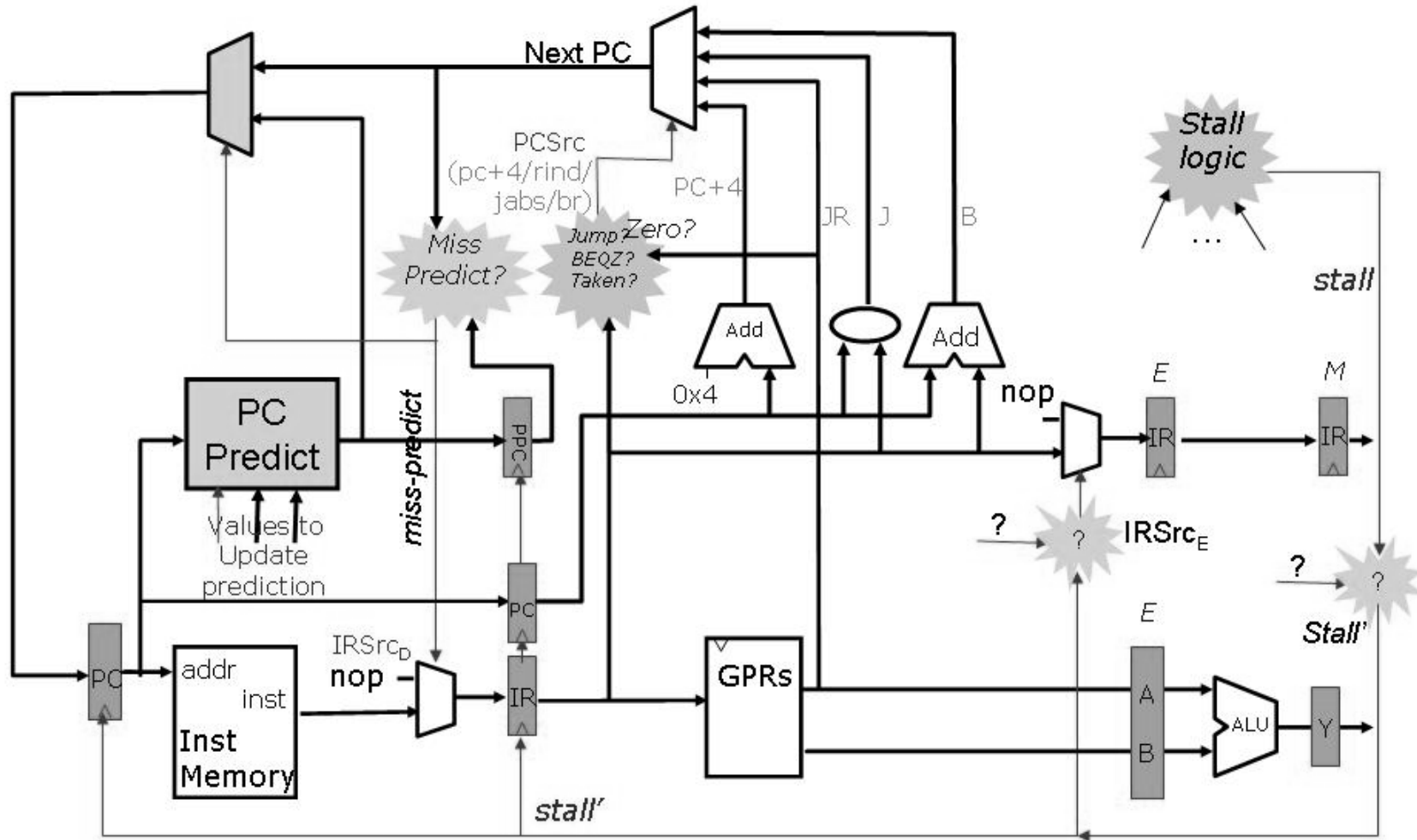


Figure M3.8-B. 5 stage MIPS pipeline with a branch predictor

Problem M3.9: Fetch Pipelines

Ben is designing a deeply-pipelined single-issue in-order MIPS processor. The first half of his pipeline is as follows:

PC	PC Generation
F1	ICache Access
F2	
D1	Instruction Decode
D2	
RN	Rename/Reorder
RF	Register File Read
EX	Integer Execute

There are no branch delay slots and currently there is **no** branch prediction hardware (instructions are fetched sequentially unless the PC is redirected by a later pipeline stage). Subroutine calls use **JAL/JALR** (jump and link). These instructions write the return address (PC+4) into the link register (r31). Subroutine returns use **JR r31**. Assume that PC Generation takes a whole cycle and that you cannot bypass anything into the end of the PC Generation phase.

Problem M3.9.A

Pipelining Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction? Immediately after what pipeline stage does the processor know the subroutine return address? How many pipeline bubbles are required when executing a subroutine return?

Problem M3.9.B

Adding a BTB

Louis Reasoner suggests adding a BTB to speed up subroutine returns. Why doesn't a standard BTB work well for predicting subroutine returns?

Problem M3.9.C

Adding a Return Stack

Instead of a BTB, Ben decides to add a return stack to his processor pipeline. This return stack records the return addresses of the N most recent subroutine calls. This return stack takes no time to access (it is always presenting a return address). Explain how this return stack can speed up subroutine returns. Describe when and in which pipeline stages return addresses are pushed on and popped off the stack.

Problem M3.9.D

Return Stack Operation

Fill in the pipeline diagram below corresponding to the execution of the following code on the return stack machine:

```
A: JAL B
A+1:
A+2:
...
```

```
B: JR r31
B+1:
B+2:
...
```

Make sure to indicate the instruction that is being executed. The first two instructions are illustrated below. The crossed out stages indicate that the instruction was killed during those cycles.

instruction	time→																	
	PC	F1	F2	D1	D2	RN	RF	EX										
A																		
A+1	PC	F1	F2	D1	D2	RN	RF	EX										

Problem M3.9.E

Handling Return Address Mispredicts

If the return address prediction is wrong, how is this detected? How does the processor recover, and how many cycles are lost (relative to a correct prediction)?

Problem M3.9.F

Further Improving Performance

Describe a hardware structure that Ben could add, in addition to the return stack, to improve the performance of return instructions so that there is usually only a one-cycle pipeline bubble when executing subroutine returns (assume that the structure takes a full cycle to access).

Problem M3.10: Managing Out-of-order Execution

This problem investigates the operation of a superscalar processor with branch prediction, register renaming, and out-of-order execution. The processor holds all data values in a **physical register file**, and uses a **rename table** to map from architectural to physical register names. A **free list** is used to track which physical registers are available for use. A **reorder buffer (ROB)** contains the bookkeeping information for managing the out-of-order execution (but, it does not contain any register data values). These components operate as described in Lecture 14 (2005).

When a branch instruction is encountered, the processor predicts the outcome and takes a snapshot of the rename table. If a misprediction is detected when the branch instruction later executes, the processor recovers by flushing the incorrect instructions from the ROB, rolling back the “next available” pointer, updating the free list, and restoring the earlier rename table snapshot.

We will investigate the execution of the following code sequence (assume that there is **no** branch-delay slot):

```
loop:  lw    r1, 0(r2)    # load r1 from address in r2
       addi  r2, r2, 4    # increment r2 pointer
       beqz  r1, skip    # branch to "skip" if r1 is 0
       addi  r3, r3, 1    # increment r3
skip:  bne   r2, r4, loop # loop until r2 equals r4
```

The diagram for Question M3.10.A on the next page shows the state of the processor during execution of the given code sequence. An instance of each instruction in the loop has been issued into the ROB (the beqz instruction has been predicted not-taken), but none of the instructions have begun execution. In the diagram, old values which are no longer valid are shown in the following format: $\overline{P4}$. The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

Problem M3.10.A

Assume that the following events occur in order (though not necessarily in a single cycle):

Step 1. The first three instructions from the next loop iteration (lw, addi, beqz) are written into the ROB (note that the bne instruction has been predicted taken).

Step 2. All instructions which are ready after Step 1 execute, write their result to the physical register file, and update the ROB. Note that this step only occurs **once**.

Step 3. As many instructions as possible commit.

Update the diagram below to reflect the processor state after these events have occurred. Cross out any entries which are no longer valid. Note that the “ex” field should be marked when an instruction executes, and the “use” field should be cleared when it commits. Be sure to update the “next to commit” and “next available” pointers. If the load executes, assume that the data value it retrieves is 0.

Rename Table			
R1	P1	P4	
R2	P2	P5	
R3	P3	P6	
R4	P0		

Physical Regs		
P0	8016	p
P1	6823	p
P2	8000	p
P3	7	p
P4		
P5		
P6		
P7		
P8		
P9		

Free List
P4
P5
P6
P7
P8
P9
⋮

Reorder Buffer (ROB)

	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit →	x		lw	p	P2			r1	P1	P4
	x		addi	p	P2			r2	P2	P5
	x		beqz		P4					
	x		addi	p	P3			r3	P3	P6
next available →	x		bne		P5	p	P0			

Problem M3.10.B

Assume that after the events from Question M3.10.A have occurred, the following events occur in order:

- Step 1.** The processor detects that the beqz instruction has mispredicted the branch outcome, and recovery action is taken to repair the processor state.
- Step 2.** The beqz instruction commits.
- Step 3.** The correct next instruction is fetched and is written into the ROB.

Fill in the diagram below to reflect the processor state after these events have occurred. Although you are not given the rename table snapshot, you should be able to deduce the necessary information from the diagram from Question M3.10.A. You do not need to show invalid entries in the diagram, but be sure to **fill in all fields** which have valid data, and update the “next to commit” and “next available” pointers. Also make sure that the **free list** contains all available registers.

Rename Table			
R1			
R2			
R3			
R4			

Physical Regs		
P0		
P1		
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		

Free List
⋮

Reorder Buffer (ROB)

	use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
next to commit										
next available										

Problem M3.10.C

Consider, (1) a single-issue, in-order processor with no branch prediction, and (2) a multiple-issue, out-of-order processor with branch prediction. Assume that both processors have the same clock frequency. Consider how fast the given loop executes on each processor, assuming that it executes for many iterations.

Under what conditions, if any, might the loop execute at a faster rate on the in-order processor compared to the out-of-order processor?

Under what conditions, if any, might the loop execute at a faster rate on the out-of-order processor compared to the in-order processor?

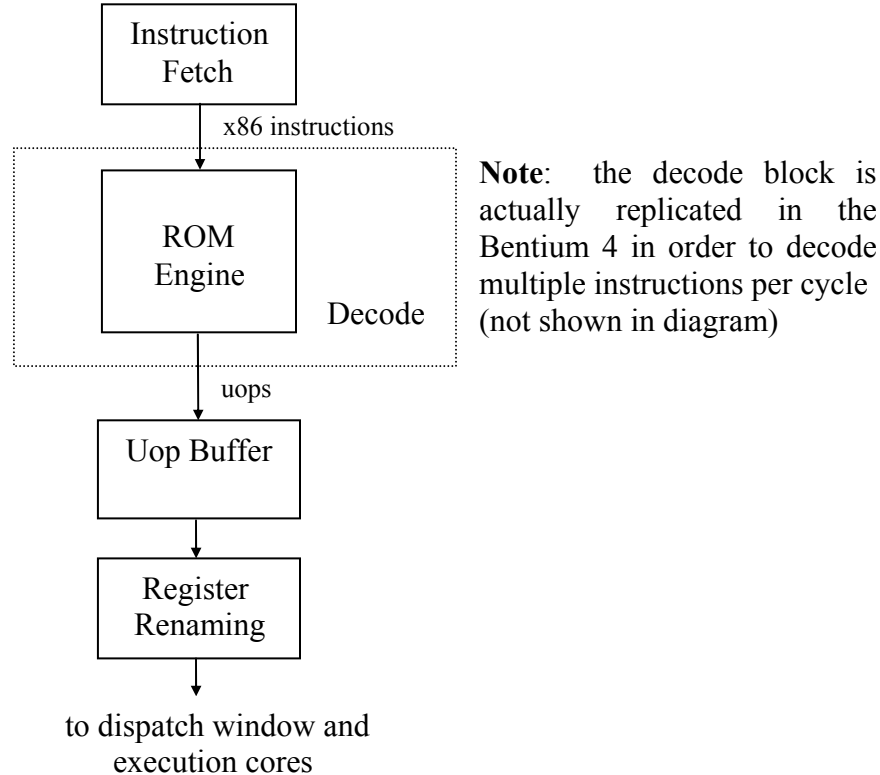
Problem M3.11: Exceptions and Register Renaming

Ben Bitdiddle has decided to start Bentel Corporation, a company specializing in high-end x86 processors to compete with Intel. His latest project is the Pentium 4, a superscalar out-of-order processor with register renaming and speculative execution.

The Pentium 4 has 8 architectural registers (EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI). In addition, the processor provides 8 internal registers T0-T7 not visible to the ISA that can be used to hold intermediary values used by microoperations (uops) generated by the microcode engine. The microcode engine is the decode unit and is used to generate uops for all the x86 instructions. For example, the following register-memory x86 instruction might be translated into the following RISC-like uops:

$$\text{ADD } R_d, R_a, \text{offset}(R_b) \rightarrow \begin{array}{l} \text{LW } T0, \text{offset}(R_b) \\ \text{ADD } R_d, R_a, T0 \end{array}$$

All 16 uop-visible registers are renamed by the register allocation table (RAT) into a set of physical registers (P0-Pn) as described in Lecture 14 (2005). There is a separate shadow map structure that takes a snapshot of the RAT on a speculative branch in case of a misprediction. The block diagram for the front-end of the Pentium 4 is shown below:



Problem M3.11.A

Recovering from Exceptions

For the Pentium 4, if an x86 instruction takes an exception before it is committed, the machine state is reset back to the precise state that existed right before the excepting instruction started executing. This instruction is then re-executed after the exception is handled. Ben proposes that the shadow map structure used for speculative branches can also be used to recover a precise state in the event of an exception. Specify an implementable strategy for taking the least number of snapshots of the RAT that would still allow the Pentium 4 to implement precise exception handling.

Problem M3.11.B

Minimizing Snapshots

Ben further states that the shadow map structure does not need to take a snapshot of all the registers in the Pentium 4 to be able to recover from an exception. Is Ben correct or not? If so, state which registers do not need to be recorded and explain why they are not necessary, or explain why all the registers are necessary in the snapshot.

Problem M3.11.C

Renaming Registers

Assume the Pentium 4 has the same register renaming scheme as the Pentium 4, as described in Lecture 14. What is the minimum number of physical registers (P) that the Pentium 4 must have to allow register renaming to work? Explain your answer.